# **Kubernetes Pod Autoscaling: FastAPI + Horizontal Pod Autoscaler (HPA)**

This document provides a comprehensive guide to deploying a web application on Kubernetes and configuring **Horizontal Pod Autoscaling (HPA)**. This project offers a practical, hands-on demonstration of how your application can automatically scale its resources up or down based on real-time demand. This dynamic scaling mechanism is crucial for ensuring optimal application performance, maintaining responsiveness under varying loads, and achieving cost efficiency by preventing over-provisioning of resources during periods of low demand.

The system utilizes a lightweight **FastAPI** application, specifically designed to simulate CPU load, which is then deployed onto a local Kubernetes cluster (Minikube). This setup allows for clear observation of HPA in action as the simulated traffic increases and decreases.

## **1. Project Objective**

The core objectives of this project are clearly defined to provide a focused learning experience:

* **Deploy a simple FastAPI web application** within a Kubernetes environment. This involves containerizing the application using Docker and then orchestrating its deployment using Kubernetes primitives. For local development and demonstration, **Minikube** is utilized to set up a single-node Kubernetes cluster.
* **Configure and observe Horizontal Pod Autoscaling (HPA)**. This is the central focus of the project. We will demonstrate how Kubernetes automatically adjusts the number of application pods (horizontal scaling) based on a predefined metric, specifically **CPU utilization**. This automated scaling mechanism is critical for ensuring that your application can efficiently handle varying traffic loads without requiring manual intervention, thereby improving both reliability and resource utilization.

By achieving these objectives, the project aims to provide a clear understanding of how to implement a self-adapting application infrastructure in Kubernetes, capable of responding dynamically to changes in demand.

## **2. How Horizontal Pod Autoscaling (HPA) Works**

Horizontal Pod Autoscaler (HPA) is a fundamental component of Kubernetes' self-healing and auto-scaling capabilities. It automatically adjusts the number of pod replicas within a Deployment, ReplicaSet, StatefulSet, or other scalable resources, based on observed resource utilization metrics (like CPU or memory) or custom metrics.

### **2.1 Core Principles of HPA**

The HPA operates as a control loop within the Kubernetes control plane, periodically evaluating metrics to make scaling decisions. Here's a breakdown of its operational principles:

* **Monitoring:** HPA continuously monitors specified metrics for the pods belonging to a target workload. By default, it can monitor CPU utilization and memory usage. For more advanced scenarios, it can be configured to use custom metrics or external metrics.
* **Target Thresholds:** You, as the administrator or developer, define target thresholds for these metrics. For instance, you might set a target CPU utilization of 50%. The HPA's goal is to maintain the average metric value across all target pods as close to this threshold as possible.
* **Scaling Decision Logic:**
  + **Scale Out (Increase Replicas):** If the average observed metric value across all pods **exceeds** the defined target threshold, the HPA will **increase** the number of pod replicas. The formula for calculating the desired replicas is approximately: desiredReplicas=ceil[currentReplicastimes(fraccurrentMetricValuedesiredMetricValue)] For example, if you have 2 pods and their average CPU is 80% (target 50%), then 2times(80/50)=2times1.6=3.2. Ceil it, and HPA will try to scale to 4 replicas.
  + **Scale In (Decrease Replicas):** Conversely, if the average observed metric value **falls below** the target threshold, the HPA will **decrease** the number of pod replicas. It typically includes a "stabilization window" or "cool-down period" to prevent rapid, unnecessary flapping (frequent scaling up and down) during temporary dips in load.
* **Min and Max Replicas:** You configure a minimum and maximum number of replicas for your deployment. HPA will never scale below the minReplicas or above the maxReplicas defined in its configuration.
* **Metrics Server:** For CPU and memory-based autoscaling, Kubernetes relies on the metrics-server. This component is a cluster-wide aggregator of resource usage data. It collects CPU and memory usage metrics from the kubelet (the agent on each node) and exposes them through the Kubernetes API (metrics.k8s.io API). The HPA controller queries the metrics-server to get the necessary resource utilization data for its scaling decisions. Without the metrics-server, CPU and memory-based HPA cannot function.

### **2.2 CPU-Based Autoscaling in This Project**

In this specific project, we focus on **CPU-based autoscaling**. Our FastAPI application is intentionally designed to consume a significant amount of CPU when its / endpoint is accessed. This controlled CPU consumption serves as the trigger for the HPA to initiate scaling events. As the load test sends numerous requests, the average CPU utilization of the existing pods rises, prompting HPA to provision new pods until the CPU load per pod falls back within the desired target range, or the maxReplicas limit is reached. When the load test stops, the CPU utilization drops, and HPA will then scale down the number of pods to conserve resources.

This demonstration provides a clear, observable example of HPA's dynamic response to workload changes, highlighting its importance for efficient resource management in Kubernetes.

## **3. Setup & Run Instructions**

This section provides a detailed, step-by-step guide to deploying the application, configuring HPA, and observing the autoscaling behavior on your local machine using Minikube.

### **3.1 Prerequisites**

Before you begin, ensure that your development environment is set up with the following tools:

* **Minikube:** A tool designed to run a single-node Kubernetes cluster directly on your local machine. It's excellent for development and testing Kubernetes features.
  + **Installation Guide:**<https://minikube.sigs.k8s.io/docs/start/>
* **Kubectl:** The official Kubernetes command-line tool. It allows you to run commands against Kubernetes clusters, deploy applications, inspect and manage cluster resources, and view logs.
  + **Installation Guide:**<https://kubernetes.io/docs/tasks/tools/install-kubectl/>
* **Docker:** A platform for developing, shipping, and running applications in containers. Minikube often uses Docker as its underlying driver to create and manage the Kubernetes node.
* **Python 3.8+:** Required for executing the load\_test.py script, which simulates traffic to trigger autoscaling, and for the FastAPI application itself.

### **3.2 Step-by-Step Guide**

Follow these steps precisely to get the system up and running:

**Start Minikube and Set Docker Environment:** First, initiate your local Kubernetes cluster using Minikube. After starting the cluster, it's crucial to configure your local Docker daemon to use Minikube's Docker environment. This step is vital because it ensures that any Docker images you build locally will be available directly within the Minikube cluster's Docker daemon, meaning Kubernetes can find and use them without needing to push them to an external registry.  
Bash  
minikube start

# Wait for Minikube to fully start and report that kubectl is configured.

# For Linux/macOS:

eval $(minikube -p minikube docker-env)

# For Windows (PowerShell):

minikube -p minikube docker-env | Invoke-Expression

* + The eval $(minikube -p minikube docker-env) (or minikube -p minikube docker-env | Invoke-Expression for PowerShell) command sets environment variables in your current terminal session that direct your Docker client to communicate with the Docker daemon running inside the Minikube VM/container.

**Build Docker Image inside Minikube:** Now, navigate to the root directory of your project (Kubernetes-Pod-Scaling-K8s-HPA-/). Build the Docker image for your FastAPI application. Because you've set your Docker environment to Minikube in the previous step, this image will be built directly into Minikube's internal Docker daemon, making it immediately accessible to Kubernetes pods.  
Bash  
docker build -t fastapi-cpu-app ./app

* + You should observe output in your terminal indicating the Docker build process, showing layers being cached or created. The final output should confirm the image fastapi-cpu-app has been successfully tagged.

**Apply Kubernetes Resources:** With the Docker image built, you can now deploy your application and configure HPA using the provided Kubernetes YAML manifest files. These files define the desired state of your application and its scaling behavior within the cluster.  
Bash  
# Deploy the FastAPI application as a Kubernetes Deployment

kubectl apply -f k8s/deployment.yaml

# Create a Kubernetes Service to expose the FastAPI app

kubectl apply -f k8s/service.yaml

# Enable the Kubernetes Metrics Server (essential for HPA to get CPU/memory metrics)

minikube addons enable metrics-server

# Apply the Horizontal Pod Autoscaler configuration

kubectl apply -f k8s/hpa.yaml

* + You will see confirmation messages for each kubectl apply command, such as deployment.apps/fastapi-cpu-app-deployment created, service/fastapi-service created, metrics-server enabled, and horizontalpodautoscaler.autoscaling/fastapi-hpa created.
  + **Note on Metrics Server:** The metrics-server is crucial. HPA relies on it to collect actual CPU and memory usage data from your running pods. Without it, HPA cannot make informed scaling decisions based on resource utilization.

## **4. Run the Application & Trigger Autoscaling**

With all Kubernetes resources deployed, you are now ready to observe the autoscaling in action.

**Get the Service URL:** To access your FastAPI application from your local machine, you need to find the external URL exposed by Minikube for your fastapi-service.  
Bash  
minikube service fastapi-service --url

* + This command will output a URL similar to http://192.168.49.2:30000. Copy this URL, as you'll use it for the load test.

**Access the Application:** Open a web browser and paste the URL obtained in the previous step. You should see a simple JSON response:  
JSON  
{"message": "CPU Load generated!"}

* + This confirms that your FastAPI application is running correctly within Kubernetes and is accessible. It also indicates that each access to this endpoint will intentionally generate CPU load on the serving pod.

**Trigger Autoscaling with a Load Test:** To initiate the autoscaling process, we need to generate significant, sustained traffic to the FastAPI application. This will increase the CPU utilization of the pods beyond the HPA's target threshold. Open a **new terminal window** (keep the previous terminals running your Minikube processes) and run the provided load testing script:  
Bash  
python load\_test.py

* + This Python script will continuously send a high volume of HTTP requests to your FastAPI service's URL, effectively stressing its CPU. The script is designed to simulate a sudden surge in demand.

**Monitor Scaling in Real-Time:** Open **another new terminal window** (you'll now have at least four terminals open: Minikube, Load Test, hpa watch, pods watch). In this new terminal, you can continuously observe the Horizontal Pod Autoscaler's status and the current number of pods in your deployment.  
Bash  
# Watch the Horizontal Pod Autoscaler status

kubectl get hpa -w

# Watch the current pods in your deployment

kubectl get pods -w

**Initial State:** When you first run kubectl get hpa -w, you will typically see output similar to:  
NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

fastapi-hpa Deployment/fastapi-cpu-app-deployment 0%/50% 1 10 1 <age>

* + The TARGETS column initially might show 0%/50% or a very low percentage, indicating low CPU utilization, and REPLICAS will be 1 (your minimum configured).
  + **Scaling Up:** As the load\_test.py script generates traffic, you will observe the TARGETS (CPU utilization percentage) for fastapi-hpa rapidly increasing (e.g., to 150%/50%, 200%/50%, etc.).
    - Shortly after the TARGETS significantly exceed the 50% threshold, Kubernetes will automatically create new pods. You will see the REPLICAS count increase (e.g., from 1 to 2, then 3, and so on, up to the configured maximum of 10).
    - Simultaneously, in the kubectl get pods -w output, you'll see new pods appearing with ContainerCreating or Running statuses.
  + **Scaling Down:** Once the load test finishes or if you manually stop the load\_test.py script, the CPU utilization across your pods will naturally decrease. After a brief "cool-down period" (a built-in delay in HPA to prevent rapid scaling fluctuations), you will observe the HPA automatically reducing the number of pods back towards the minReplicas count (1 in k8s/deployment.yaml). The kubectl get pods -w output will show pods terminating.

This real-time observation provides a clear and compelling demonstration of Kubernetes HPA effectively managing your application's resource allocation based on demand.

## **5. Folder Structure**

The project is structured logically to facilitate understanding, development, and deployment of the application and its Kubernetes resources.

k8s-autoscale-project/

├── app/ # Contains the FastAPI application and its Dockerfile

│ ├── main.py # FastAPI application: defines an endpoint that responds with a message and intentionally generates CPU load.

│ └── Dockerfile # Dockerfile to containerize the FastAPI application into a portable image.

├── k8s/ # Kubernetes manifest files for deploying and managing the application

│ ├── deployment.yaml # Defines the Kubernetes Deployment for the FastAPI app, specifying desired replicas, container image, resource requests/limits.

│ ├── service.yaml # Defines the Kubernetes Service to expose the FastAPI app within the cluster and potentially externally (NodePort in this case).

│ └── hpa.yaml # Defines the Horizontal Pod Autoscaler configuration, specifying the target deployment, min/max replicas, and target CPU utilization.

├── load\_test.py # Python script designed to simulate high traffic by sending continuous requests to the FastAPI service to trigger autoscaling.

├── README.md # This comprehensive project documentation (you are currently reading this).

└── SUMMARY.md # A brief, high-level overview of the project.

* **app/**: This directory contains the source code for the FastAPI application.
  + main.py: This Python file implements the FastAPI web application. It includes an endpoint that, when accessed, consumes CPU resources to simulate a workload, making it suitable for HPA demonstration.
  + Dockerfile: This file specifies the instructions for building a Docker image of the FastAPI application.
* **k8s/**: This directory holds all the Kubernetes manifest files (.yaml) required to deploy and manage the application within a Kubernetes cluster.
  + deployment.yaml: Describes a Kubernetes Deployment object. It defines how the FastAPI application pods should be created, updated, and scaled, including the Docker image to use, the number of initial replicas, and resource requests/limits for the containers. Resource requests are critical for HPA to accurately calculate CPU utilization.
  + service.yaml: Defines a Kubernetes Service object. This creates a stable network endpoint for the FastAPI application, allowing other services inside the cluster or external clients to access it, abstracting away the dynamic IP addresses of individual pods.
  + hpa.yaml: Defines a HorizontalPodAutoscaler resource. This is where the autoscaling logic is configured: it specifies which Deployment to scale, the minimum and maximum number of pods, and the target average CPU utilization percentage that HPA should try to maintain across all pods.
* **load\_test.py**: This stand-alone Python script is used to generate a sustained high volume of HTTP requests to the deployed FastAPI service. Its purpose is to create enough CPU load on the application pods to trigger the HPA's scaling-out behavior.
* **README.md**: The primary documentation file for the project, providing setup instructions, architectural overview, and usage details.
* **SUMMARY.md**: A concise, high-level summary of the project's purpose and components.

This structured approach ensures that each aspect of the project is clearly separated, making it easy to understand, modify, and manage.

## **6. Features Demonstrated**

This project effectively serves as a practical demonstration of several key Kubernetes and cloud-native concepts, making it an excellent learning resource for anyone interested in application scaling and orchestration:

* **Dockerized FastAPI Application:** It showcases how to package a simple yet functional Python web application (FastAPI) into a Docker image. This process ensures application consistency and portability across different environments, a cornerstone of modern cloud-native development.
* **Kubernetes Deployment:** The project provides a clear example of correctly defining and deploying an application using Kubernetes Deployment resources. This includes setting up desired replica counts, specifying container images, and defining resource requests and limits, which are essential for efficient scheduling and autoscaling.
* **Kubernetes Service:** It demonstrates how to expose a deployed application within the cluster and to external traffic by defining a Kubernetes Service. In this specific example, a NodePort service is used to make the application accessible from outside the Minikube cluster.
* **CPU-Based Horizontal Pod Autoscaling (HPA):** This is the central feature. The project provides a practical implementation and observable demonstration of HPA, illustrating how Kubernetes automatically scales the number of pods (from a minimum of 1 up to a maximum of 10 replicas) based on the observed average CPU utilization across those pods.
* **Metrics Server Integration:** It highlights the necessity of the metrics-server addon in a Kubernetes cluster for HPA to function correctly. The metrics-server is responsible for collecting and serving resource usage data (like CPU and memory) from nodes and pods, which HPA then consumes to make scaling decisions.
* **Real Traffic Simulation:** The inclusion of a load\_test.py Python script is crucial. It simulates realistic load by generating a high volume of requests, allowing users to actively observe the dynamic response of the autoscaling mechanism in real-time, just as it would in a production environment under varying traffic conditions.

By integrating these features, the project offers a comprehensive and tangible understanding of how modern applications can achieve resilience, performance, and cost-efficiency in a Kubernetes environment through intelligent autoscaling.

## **7. Contributing**

Contributions to this project are highly encouraged and welcome! Whether you have suggestions for improvements, ideas for new features, or find a bug that needs fixing, your input is valuable.

To contribute:

1. **Open an Issue:** If you identify a bug, have a feature request, or want to propose an enhancement, please open an issue on the GitHub repository. Provide a clear and detailed description, including steps to reproduce any bugs and explaining the rationale behind your feature ideas.
2. **Submit a Pull Request:** If you've implemented a solution, added a new feature, or made improvements to the existing codebase, please feel free to submit a pull request. Ensure your code adheres to any existing coding styles, is well-commented, and includes appropriate tests if applicable.

Your efforts help make this project more robust, feature-rich, and helpful for the broader community.

## **8. License**

This project is licensed under the **MIT License**.

The MIT License is a very permissive open-source license. It allows you to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software. The only condition is that the original copyright notice and permission notice must be included in all copies or substantial portions of the software.

For the full text of the license, please refer to the LICENSE file located in the root directory of this project.

## **9. Acknowledgments**

We extend our sincere appreciation to the communities and teams behind the following exceptional open-source projects. Their dedication and the powerful tools they provide have been fundamental in the creation and demonstration of this project:

* **The Kubernetes and Minikube communities:** For developing and maintaining such robust and transformative container orchestration tools that empower developers to deploy and manage applications at scale.
* **The FastAPI team:** For creating an outstanding, high-performance, and incredibly developer-friendly web framework that simplifies API development in Python.
* **Everyone contributing to open-source software:** The vast ecosystem of open-source projects is the foundation upon which modern technology is built. We are grateful for all the collective efforts that make projects like this possible.